Caminhamento Paralelo Barnes-Hut com Vetorização AVX2

Armando L. N. Delgado[1], **Rodrigo Morante**[1], **Wagner M. Nunan Zola**[1],

[1] Departamento de Informática
Universidade Federal do Paraná (UFPR) – Curitiba, PR – Brazil

nicolui@inf.ufpr.br, rodrigomorante@gmail.com, wagner@inf.ufpr.br

Abstract. *The Barnes-Hut algorithm is an approximate method used for gravitational simulation of N-Bodies. The irregular nature of this code presents challenges for its computing in parallel systems. Additional obstacles in this computation pattern arise to effectively use the computational capability of multicore architectures with SIMD instructions. Our focus on this work is to implement and analyze the efficiency of the Barnes-Hut parallel traversals with implicit octrees and the use of AVX2 vector instructions. The experiments validate the effectiveness of our method, which has high GFLOP/s rate and takes considerably less energy in simulations.*

Resumo. *O algoritmo Barnes-Hut é um método aproximado amplamente usado na simulação gravitacional de N-Corpos. A natureza irregular desse código apresenta desafios para sua computação em sistemas paralelos. Obstáculos adicionais ocorrem nesse padrão de computação quando se deseja a utilização eficaz da capacidade computacional de arquiteturas* multicore *com instruçoes SIMD. O enfoque deste trabalho é implementar e analisar a eficiência do caminhamento paralelo Barnes-Hut com* octrees *implícitas e uso de instruções vetoriais AVX2. Os experimentos demonstram a efetividade do método, que apresenta altas taxas de GFLOP/s e economia de energia nas simulações.*

1. Introdução

A simulação dos N-corpos gravitacional pelo método direto (partícula-partícula), embora exata, apresenta complexidade quadrática $\mathcal{O}(N^2)$ tornando inviável seu uso em larga escala. No entanto, é importante o estudo de técnicas eficientes para implementação dos métodos diretos [Arora et al. 2009] em arquiteturas *multicore* e aceleradores, pois estes podem ser também utilizados na composição de outros métodos que apresentem menor complexidade computacional.

Barnes e Hut [Barnes and Hut 1986] propuseram um algoritmo aproximado (*BH*) de cálculo de forças de complexidade $\mathcal{O}(N \log N)$ baseado no caminhamento em *octrees*. O algoritmo exibe padrões comuns de computação e comunicação de importância no estudo de modelos e arquiteturas de programação paralela [Asanovic et al. 2006].

No algoritmo *BH*, grupos de partículas são organizados em subárvores, com um nodo pai resumindo as propriedades desse cluster de partículas. *Octrees* são estruturas de dados de subdivisão de espaço que reduzem a complexidade computacional do algoritmo de cálculo de força. O processamento de estruturas de dados esparsas, como *octrees*, apresenta problemas de desempenho para aplicação de paralelismo bem como na utilização de instruções SIMD (*Single Instruction, Multiple Data*) em arquiteturas *multicore* e particularmente em GPUs, devido ao padrão irregular de acesso aos dados. Comumente são utilizados formatos compactos ou estruturas de aceleração especiais para representação de dados esparsos, minimizando problemas com os padrões irregulares de acesso.

O trabalho em [Nunan Zola et al. 2014] apresentou o uso de *octrees* implícitas que possibilitam o caminhamento eficiente Barnes-Hut em GPUs. No presente trabalho investigamos a aplicabilidade de tais estruturas no método *BH* e vetorização em AVX2 (*Advanced Vector Extensions*, da Intel) para simulação paralela em arquiteturas *multicore*, demonstrando sua eficácia. Cabe observar que o caminhamento na árvore apresenta padrões não regulares de computação, mesmo com a utilização de *octrees* implícitas. Assim, a aplicação de vetorização automática pela simples utilização de opções de compilação não é eficaz, razão pela qual nesse trabalho a vetorização é codificada explicitamente com o uso de instruções intrínsecas AVX2.

O restante do trabalho está organizado da seguinte forma: Na Seção 2 analisamos trabalhos relacionados. Na Seção 3 apresentamos com mais detalhes o método Barnes-Hut. Na Seção 4 é explicada nossa implementação paralela deste método, bem como o uso de instruções AVX2 para a melhoria do desempenho da simulação. Na Seção 5 os resultados obtidos são apresentados. Finalizamos com as conclusões na Seção 6.

2. Trabalhos relacionados

Métodos eficientes foram desenvolvidos em [Burtscher and Pingali 2011] para gerar e percorrer *octrees* baseados em ponteiros em GPUs em simulações gravitacionais de *BH*. O desempenho deste código foi analisado por [Zecena et al. 2013], incluindo comparações com códigos para CPUs *multicore*. Outro simulador que é executado em GPUs foi apresentado por [Bédorf et al. 2012]. Este código utiliza uma *octree* esparsa baseada em ponteiros, mas com uma estrutura diferente. As árvores são geradas e caminhadas em pré-ordem. Os corpos são ordenados segundo suas chaves de Morton. Esta etapa e as subsequentes onde existem transferências de dados para reordenação dos nodos internos ocupa aproximadamente 66% do tempo da geração da árvore. Uma possível vantagem de ordenar as folhas segundo a ordem de Morton é que melhora a eficiência do cache durante o caminhamento na árvore.

O uso de *octrees* implícitas no caminhamento *BH* em GPUs foi apre-

sentado em [Nunan Zola et al. 2014]. A fase de transformação da *octree* esparsa também inclui o movimento de dados para o *layout* implícito. A transformação da árvore coloca os nodos em *pré-ordem*, usando a *octree* baseada em ponteiros construída inicialmente. Essa ordem é equivalente à ordem de Morton de todos os nodos da árvore. Desta forma, a etapa de caminhamento e cálculo de forças apresentou considerável ganho de desempenho em relação a outras implementaçoes em GPU.

O uso de paralelismo aliado à vetorização SIMD/AVX2 foi explorado em diversos trabalhos envolvendo problemas de N-corpos e caminhamento em árvores [Yokota 2012, Long Wang et al. 2015, Arora et al. 2009], especialmente onde havia cálculo direto de forças gravitacionais. Assim é o caso em [Lange and Fortin 2014] que usa o método *dual tree* e que não faz uso de instruções SIMD intrínsecas diretamente, deixando a cargo do compilador fazer o trabalho de vetorização. Isso é possível principalmente porque nesses métodos as interações são do tipo célula-célula (C-C) o que facilita a vetorização automática. As interações C-C equivalem a aplicar o método direto entre corpos internos às células que, nesse ponto, é um padrão regular de computação.

3. Simulação Barnes-Hut

O *método Barnes-Hut* particiona o espaço utilizando uma *octree*, reduzindo assim o número de cálculos no caminhamento. Sejam l o lado de uma célula (cúbica), d a distância entre o corpo de referência e o baricentro da célula e δ a distância entre o centroide e o baricentro da célula. Seja também θ o *parâmetro de abertura* (Figura 1, esquerda). O objeto e a célula estão suficientemente afastados quando se cumpre o critério da Figura 1 à direita: neste caso o baricentro dos corpos dentro da célula é utilizado para calcular a força total sobre o corpo de referência. Senão, a célula é percorrida recursivamente e todas as células nela contidas são testadas, eventualmente sendo utilizados os corpos dentro delas para calcular a força. A versão da equação sem raízes quadradas diminui o número de cálculos. Além disso, o lado direito da equação é denominado COV (*Cell Opening Value*) e pode ser pré-computado e armazenado para cada nodo interno da árvore. Fazendo $\theta = 0$ todas as células são abertas, obtendo-se os mesmos resultados que com a versão quadrática. O método *BH* baseado em árvores tem complexidade $\mathcal{O}(N \log N)$.

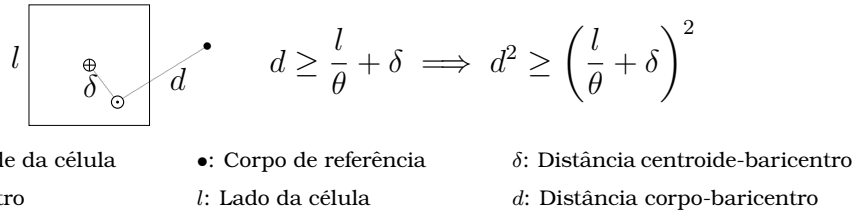

Figura 1. Critério para abertura de célula.

4. Caminhamento Barnes-Hut em árvore implícita com vetorização SIMD/AVX2

Uma versão paralela do método Barnes-Hut que usa ávores esparsas foi implementada em C++ com uso de paralelismo em `pthreads`. O estado inicial dos corpos é gerado seguindo o modelo de Plummer para aglomerados globulares [Plummer 1911]. A simulação consiste em ordenar os corpos segundo suas chaves de Morton e gerar uma *octree* esparsa com todos os corpos, para em seguida calcular as forças exercidas sobre cada corpo com uso do caminhamento na árvore gerada. Na execução com múltiplas *threads*, cada uma destas atua calculando forças sobre um subconjunto dos corpos, com caminhamento na árvore diferenciado por *core* e preenchimento dos níveis de cache correspondentes a esse *core*. Consideramos nos experimentos apenas a execução de um fluxo de execução por *core* ou por *hyperthread*.

Foi também desenvolvida uma variante deste algoritmo, na qual, a *octree* esparsa gerada é transformada em *octree* implícita, conforme mostrado na Figura 2. Na presente versão, a geração e a transformaçâo são sequenciais. Nesta *octree* os dados dos baricentros (massa, posição e raio) e das folhas (massa e posição), assim como os ponteiros a nodos internos e folhas vizinhos são armazenados em um arranjo. O cálculo das forças nessa versão, e a atualização das velocidades e posições dos corpos é feito na árvore implícita mediante o uso de paralelismo vetorial com AVX2 codificado manualmente com instruções intrínsecas, além do paralelismo de *threads*.

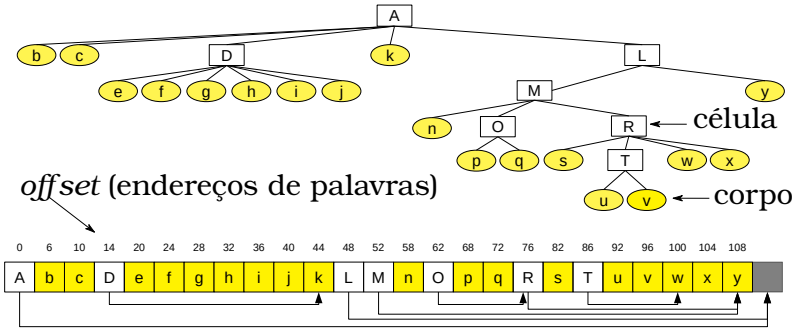

Figura 2. *Octree* esparsa e equivalente implícita. *Offsets* são deslocamentos a partir do primeiro elemento do arranjo.

Deve-se ressaltar que o peso maior nos tempos de simulação recai no cálculo das forças, ou seja, a etapa durante a qual a *octree* é percorrida. A ordem dos corpos na *octree* implícita segue a ordem de Morton. No entanto, a ordenação nesse caso é feita somente com pares (chave de Morton, valor) onde o valor é o índice da posição original no vetor de corpos de referência. Esses valores são armazenados em um vetor de índices que faz o mapeamento dos corpos na árvore para suas posições no vetor de corpos de referência.

O algoritmo proposto em [Nunan Zola et al. 2014] para o caminhamento na árvore implícita passa a ser implementado para ser executado em CPU *multicore*, usando-se explicitamente instruções intrínsecas

SIMD/AVX2, pois as estruturas de computação não-regulares existentes neste algoritmo torna insuficiente a simples utilização de opções do compilador usado (GCC) para otimização para AVX2. Durante o caminhamento na árvore implícita, para cada corpo de referência buscado via vetor de índices de mapeamento procede-se com o cálculo das distâncias entre partículas e células, recíprocos de distâncias, comparações entre distâncias e valores de abertura de células e o cálculo de acelerações e forças, usando-se instruções intrínsecas AVX2 para operações aritméticas, *gathering* e *load*.

Com o uso de vetorização, os cálculos de força são realizadas de forma que a recuperação dos valores dos vetores de posição e aceleração, e posterior uso no cálculo é paralelizado nos registradores AVX2. Com isto, as operações são realizadas em grupos de 8 (tamanho de um registrador AVX2), o que aumenta o volume de operações por interação partícula-partícula (P-P) e partícula-célula (P-C) em relação a um caminhamento escalar. Esse aumento em computações é benigno, pois melhora a precisão dos cálculos. Além disso o uso da vetorização AVX2 resulta em alta para taxa de FLOP/s, diminuição nos tempos de caminhamento e no consumo de energia. A abertura de uma célula (AC) ocorre quando um corpo de referência está suficientemente próximo a esta (Figura 1). Nesse caso o algoritmo visita os nodos internos e calcula forças de interação com esses nodos. Caso não ocorra a abertura o número de operações é menor pelo cálculo aproximado pelo uso do centro de massa da célula e tomando-se o desvio da subárvore nesse ponto.

Foi também definido um mecanismo para utilização da vetorização AVX2 em conjunto com o processamento em *threads*. O processamento de corpos foi dividido por uma quantidade configurável de *threads* de acordo com o *multicore* utilizado. A utilização da *octree* implícita possibilita um balanceamento de carga praticamente ideal, e realizado de maneira simples no processador *multicore*.

5. Resultados experimentais

Para verificar a performance da execução sequencial do algoritmo de caminhamento com árvore implícita e codificação explícita da vetorização AVX2 utilizou-se um processador Kabylake Intel Core i7-7500U a 2.70GHz (2 *cores*, 2 *threads*/*core*), 8 GiB de memória RAM. Este é um ambiente dedicado para permitir o acesso da ferramenta `likwid` [Treibig et al. 2010] aos contadores de performance do processador, usados para obter as medidas de FLOP/s, energia e uso do cache L1. Para verificação da escalabilidade com mais núcleos de processamento e diversas *threads* foi utilizado com processador Broadwell Intel Xeon E5-2686v4 a 2.30GHz (8 *cores*, 2 *threads*/*core*), 125 GiB de memória RAM. Nos dois casos o sistema Linux com *kernel* 4.15.0 foi usado.

Como parâmetros das simulações foi utilizado o valor $\theta = 0.5$ para o parâmetro de abertura de uma célula da *octree*.

Como já indicado na seção 4 a vetorização AVX2 traz um aumento de desempenho na quantidade de FLOP/s (Figura 3), embora se note um comportamento importante: esta quantidade é crescente até uma certa quantidade de partículas para depois decrescer. Isto ocorre quando a partir de um certo ponto do caminhamento a quantidade de interações com abertura de célula (AC) é maior que a interação entre corpos. Como nas interações AC não existem cálculos de forças, a quantidade de FLOP/s diminui. Outro fator importante que a *octree* implícita e posterior uso de AVX2 introduzem é a consequente diminuição de consumo de energia (Figura 3).

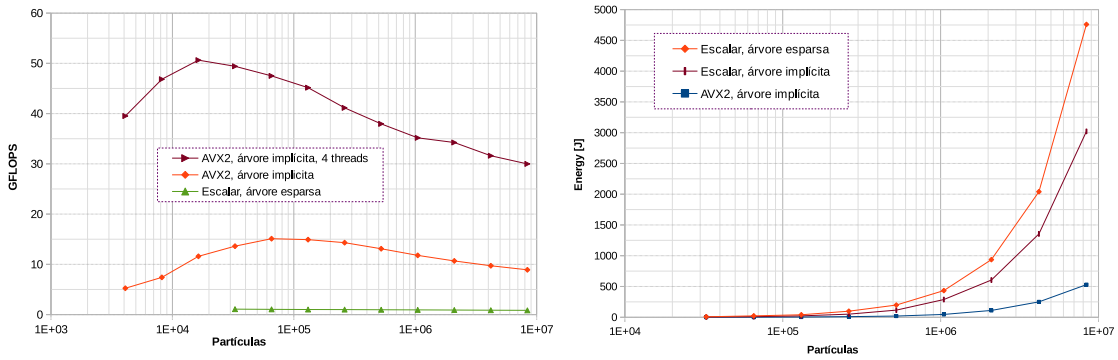

Figura 3. FLOP/s e Energia - Processador Kabylake (partículas em escala logarítmica)

A Tabela 1 mostra para o caminhamento em 8Mi corpos os valores de uso do cache L1, os tempos de caminhamento (previsto e medido), bem como o *speedup* (calculado conforme [Hennessy and Patterson 2011]), tomando como *baseline* o tempo de caminhamento escalar na árvore esparsa. O *speedup* no tempo previsto calculado em termos de CPI (clocks por instrução) confirma o valor baseado no tempo medido. Note-se que a taxa de requisição L1, que representa quantos acessos existem em média por instrução, é baixa para AVX2, o que significa que a execução destas instruções causa menos impacto na utilização do cache.

Algoritmo	Taxa requisição L1	Instruções encerradas	Mem Load acertos em L1	Mem Load faltas em L1	CPI	tempo previsto (clocks)	tempo medido (s)	Speedup previsto (1 thread)	Speedup medido (1 thread)
Octree esparsa escalar	0,220	3,06E+12	1,02E+12	3,18E+10	0,78	2,38E+12	682,62	1,00	1,00
Octree implícita escalar	0,050	1,28E+12	4,52E+10	4,52E+10	1,1	1,40E+12	403,12	1,70	1,69
Octree implícita AVX2	0,069	1,47E+11	6,95E+09	3,38E+09	1,37	2,02E+11	58,93	11,78	11,58

Tabela 1. Caminhamento *octree*: Tempos, *speedup* e uso do cache L1 - Processador Kabylake (8Mi corpos)

Para testar a escalabilidade com *threads* para as versões em árvore implícita (escalar e AVX2) em relação ao caminhamento na árvore esparsa (*baseline* com uma *thread*) simulamos uma distribuição de 8 Mi corpos no processador Broadwell para diversas quantidades de *threads*, calculando assim a aceleração (*speedup*) obtida com diferentes quantidades de *threads* (Tabela 2).

Threads	1	2	4	8	16
Tempo escalar, árvore esparsa (s)	1.318,81	682,44	351,38	184,09	149,03
Tempo AVX2, árvore implícita (s)	284,49	139,87	70,25	36,65	27,48
Speedup escalar, árvore esparsa	1,00	1,93	3,75	7,16	8,85
Speedup AVX2, árvore implícita	**4,64**	**9,43**	**18,77**	**35,98**	**48,00**

Tabela 2. Escalabilidade (*strong scaling*) - Processador Broadwell (8Mi corpos)

Ao observar o comportamento em *threads* das versões esparsa e implícita de caminhamento nas árvores (Tabela 2) nota-se vantagem na implementação AVX2, com um *speedup* de 4.64 a 48 vezes em relação à árvore esparsa e boa escalabilidade. A diferença para o *speedup* no processador Kabylake se deve às diferentes características deste, mais recente que o processador Broadwell, além de outras características diferentes dos componentes do sistema.

6. Conclusões e trabalhos futuros

Neste trabalho apresentamos uma implementação do algoritmo Barnes-Hut. A nova versão paralela gera uma *octree* em *layout* implícito e o cálculo das forças é acelerado fazendo uso de instruções AVX2. São apresentados resultados de tempo e *speedup* para várias *threads*.

O caminhamento na árvore implícita tem menor demanda de acessos em memória e executa menor número de instruções em comparação com o caminhamento em *octree* esparsa conforme se verifica na Tabela 2. A taxa de requisições ao cache L1 também é menor no caminhamento com árvores implícitas. O fato da medida de CPI ser menor no caminhamento com árvore esparsa se deve ao maior uso de instruções com números inteiros nessa versão, além de processar mais instruções escalares. As previsões de *speedup* da Tabela 1 se confirmaram na prática. Mesmo com maior CPI no caso AVX2 obtém-se menor tempo pelo uso de SIMD além de economia de energia. Constata-se que o desempenho melhora com o uso de AVX2 se comparado com a alternativa escalar e que há melhora de desempenho da versão implementada com *octree* implícita em relação ao uso da árvore esparsa.

Como metas futuras pretende-se expandir a implementação em AVX2 e paralelizar outras etapas da simulação. A versão apresentada nesse trabalho considera a escalabilidade intra-nodo de computação com uso de *pthreads*, apesar de possuir alguma capacidade de funcionamento distribuído via MPI, que não foi explorada. No entanto estamos desenvolvendo alternativas para a distribuição da árvore implicita e pretendemos testar a escalabilidade da solução em cluster.

Referências

Arora, N., Shringarpure, A., and Vuduc, R. W. (2009). Direct n-body kernels for multicore platforms. In *ICPP 2009, International Conference on Parallel Processing, Vienna, Austria, 22-25 September 2009*, pages 379–387.

Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley.

Barnes, J. and Hut, P. (1986). A hierarchical O(N log N) force-calculation algorithm. *Nature*, 324:446–449.

Bédorf, J., Gaburov, E., and Portegies Zwart, S. (2012). A sparse octree gravitational n-body code that runs entirely on the GPU processor. *J. Comput. Phys.*, 231(7):2825–2839.

Burtscher, M. and Pingali, K. (2011). Chapter 6 - an efficient CUDA implementation of the tree-based barnes hut n-body algorithm. In Hwu, W.-m. W., editor, *GPU Computing Gems Emerald Edition*, pages 75 – 92. Morgan Kaufmann, Boston.

Hennessy, J. L. and Patterson, D. A. (2011). *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition.

Lange, B. and Fortin, P. (2014). Parallel dual tree traversal on multi-core and many-core architectures for astrophysical n-body simulations. In *Euro-Par 2014: Proceedings of the 20th International European Conference on Parallel and Distributed Computing*, pages 716–727.

Long Wang, R. S. et al. (2015). NBODY6++GPU: ready for the gravitational million-body problem. *Monthly Notices of the Royal Astronomical Society*, 450(4):4070–4080.

Nunan Zola, W. M., Bona, L. C., and Silva, F. (2014). Fast GPU parallel N-Body tree traversal with Simulated Wide-Warp. In *Parallel and Distributed Systems (ICPADS), 2014 20th IEEE International Conference on*, pages 718–725.

Plummer, H. C. (1911). On the problem of distribution in globular star clusters. *Monthly Notices of the Royal Astronomical Society*, 71:460–470.

Treibig, J., Hager, G., and Wellein, G. (2010). Likwid: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA.

Yokota, R. (2012). An FMM based on dual tree traversal for many-core architectures. *CoRR*, abs/1209.3516.

Zecena, I., Burtscher, M., Jin, T., and Zong, Z. (2013). Evaluating the performance and energy efficiency of n-body codes on multi-core cpus and gpus. In *32nd IEEE International Performance Computing and Communications Conference*, IPCCC'13.